Continuations, Processes, and Sharing

Paul Downen, Luke Maurer, Zena M. Ariola, Daniele Varacca

University of Oregon, Université Paris Diderot

September 8, 2014
The plethora of semantic artifacts

▶ Many ways to understand programming languages:
  ◀ small-step semantics
  ◀ big-step semantics (natural semantics)
  ◀ abstract machines
  ◀ continuation-passing style transformations
  ◀ …

▶ Different tools; different views
  ◀ High-level reasoning
  ◀ Low-level reasoning
  ◀ Proof development
Getting along together

Q  But which one to choose?

A  All of them!

Q  But how do we know that they agree?

A  Systematic inter-derivation; correct by construction (Danvy et al.)
Getting along together

Q But which one to choose?

A All of them!
Getting along together

Q  But which one to choose?

A  All of them!

Q  But how do we know that they agree?

Systematic inter-derivation; correct by construction (Danvy et al.)
Getting along together

Q  But which one to choose?
A  All of them!

Q  But how do we know that they agree?
A  Systematic inter-derivation; correct by construction (Danvy et al.)
Processes as a semantic tool

- Embedding into processes ($\pi$-calculus)
- Computation as communication
- Strong resemblance to continuation-passing
Processes as a semantic tool

- Embedding into processes ($\pi$-calculus)
- Computation as communication
- Strong resemblance to continuation-passing
What do processes have to offer?

- Some computations more direct in $\pi$- than $\lambda$-calculus
  - Concurrency
  - Non-determinism
  - Change over time

- Simple story for memoization

- Reveals techniques in implementations of lazy languages
  - “Black holes” in GHC
From continuations to processes
Example: function composition

Transform the function composition

\[ f (g \ 1) \]

into a process
Result-named style

Name intermediate results of serious computations:

\[ f \ (g \ 1) \]

goes to

\[ \text{let } y = g \ 1 \ \text{in} \ f \ y \]
Rewrite into continuations:

```markdown
let y = g 1 in f y
```

goes to

```markdown
\text{g}(1, (\lambda y. f(y, \text{ret})))
```
Value-named style

Name all serious values:

\[ g(1, (\lambda y. f(y, \text{ret}))) \]

goes to

\[ \text{let } k = \lambda y. f(y, \text{ret}) \text{ in } g(1, k) \]
Environment-based CPS

Rewrite into explicit environment:

\[
\text{let } k = \lambda y. f(y, \text{ret}) \text{ in } g(1, k)
\]

goes to

\[
\nu k. k := \lambda y. f(y, \text{ret}) \text{ in } g(1, k)
\]
Process encoding

Rewrite into processes:

\[ \nu k. k := \lambda y. f(y, \text{ret}) \text{ in } g(1, k) \]

goes to

\[ \nu k (\! k(y). \bar{f} \langle y, \text{ret} \rangle \mid \bar{g} \langle 1, k \rangle) \]
Uniform CPS transform (CBN and CBV)

\[ C[x] \triangleq \lambda k. x \ k \]
\[ C[\lambda x. M] \triangleq \lambda k. k (\lambda (x, k'). C[M] k') \]

**Call-by-name**

\[ C[MN] \triangleq \lambda k. C[M](\lambda v. v (\lambda k'. C[N] k', k)) \]

**Call-by-value**

\[ C[MN] \triangleq \lambda k. C[M](\lambda v. C[N] (\lambda w. v (\lambda k'. k' w, k))) \]
Uniform CPS to Uniform $\pi$-encoding

\[ C[x]k = x \; k \]
\[ \mathcal{N} \circ C[x]k = x \; k \]
\[ \mathcal{P} \circ \mathcal{N} \circ C[x]k = \bar{x}\langle k \rangle \]

\[ C[\lambda x. M]k = k \; (\lambda(x, k'). C[M]k') \]
\[ \mathcal{N} \circ C[\lambda x. M]k = \nu f. f := \lambda(x, k'). \mathcal{N} \circ C[M]k' \text{ in } k f \]
\[ \mathcal{P} \circ \mathcal{N} \circ C[\lambda x. M]k = \nu f \; (\!f(x, k'). \mathcal{P} \circ \mathcal{N} \circ C[M]k' \mid \bar{k}\langle f \rangle) \]

\[ \ldots \]
Interlude: Of variables and values
A mismatch

Soundness: steps in source are steps in target

\[(\lambda x. \lambda y. y)z = \lambda y. y\]

This is invalid by CBV transform of application:

\[C[(\lambda x. \lambda y. y)z] = \lambda k. z (\lambda w. C[\lambda y. y]k) \neq C[\lambda y. y]\]
Variables are not values

\[ C[x] \triangleq \lambda k. x \ k \]

In CBN we need to execute a computation
In CBV we need to lookup or fetch the value

<table>
<thead>
<tr>
<th>Plotkin CBV</th>
<th>Restricted CBV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values: ( V ::= x \mid \lambda x. M )</td>
<td>( V ::= \lambda x. M )</td>
</tr>
<tr>
<td>Evaluation Contexts: ( E ::= [] \mid EM \mid VE )</td>
<td>( E ::= [] \mid EM \mid VE )</td>
</tr>
</tbody>
</table>

\((\lambda x. M) V = M\{V/x\}\) \hspace{1cm} (\beta_v)
Correctness bisimulation
Criteria for correctness

A transformation should preserve observable results of a program:

- **Termination**: the program reaches an answer
- **Divergence**: the program loops forever
- **Getting stuck**: the program cannot proceed
How the proof should go

- $\mathcal{T}[\cdot]$ preserves immediate results
  - If $M$ is an answer then $\mathcal{T}[M]$ is an answer
  - ...

- $\mathcal{T}[\cdot]$ preserves reduction

\[
\begin{array}{c}
M \quad \xrightarrow{\downarrow} \quad \mathcal{T}[M] \\
N \quad \xrightarrow{\downarrow} \quad \mathcal{T}[N]
\end{array}
\]
How the proof actually goes

- $\mathcal{T}[]$ preserves immediate results
  - If $M$ is an answer then $\mathcal{T}[M]$ reaches an answer
  - …

- $\mathcal{T}[]$ preserves reduction

\[
\begin{array}{ccc}
M & \xrightarrow{\mathcal{T}} & N \\
\Downarrow & & \Downarrow \\
\mathcal{T}[M] & \xrightarrow{\cdot} & \mathcal{T}[N]
\end{array}
\]
How the proof actually goes

- $\mathcal{T}[]$ preserves immediate results
  - If $M$ is an answer then $\mathcal{T}[M]$ reaches an answer
  - ...

- $\mathcal{T}[]$ preserves reduction?

\[ M \longrightarrow \quad N \quad \downarrow \quad \downarrow \]
\[ \mathcal{T}[M] \quad \mathcal{T}[N] \quad \longrightarrow \quad P \]
How the proof actually goes

- $\mathcal{T}[]$ preserves immediate results
  - If $M$ is an answer then $\mathcal{T}[M]$ reaches an answer
  - ...

- $\mathcal{T}[]$ preserves reduction???
Out-of-synch computations: administration

Source:

$$(\lambda x. x)(\lambda y. y) \mapsto \lambda y. y$$

Target:

$$C[(\lambda x. x)(\lambda y. y)]\ ret \mapsto ret\ (\lambda(y, k). (\lambda k'. y\ k')\ k)$$

$$C[\lambda y. y]\ ret \mapsto ret\ (\lambda(y, k). (\lambda k'. y\ k')\ k)$$
Out-of-synch computations: aliasing

Source:

\[(\lambda f. g(f, f))(\lambda x. x) \mapsto g((\lambda x. x), (\lambda x. x))\]

Target:

\[\mathcal{N}[((\lambda f. g(f, f))(\lambda x. x))]
= \nu i. i := \lambda x. x \textbf{in} (\lambda f. g(f, f)) \ i
\mapsto \nu i. i := \lambda x. x \textbf{in} g(i, i)\]

\[\mathcal{N}[(\lambda x. x), (\lambda x. x))]
= \nu i. i := \lambda x. x \textbf{in} \nu j. j := \lambda x. x \textbf{in} g(i, j)\]
Reasoning up to out-of-synch administration

Define an administrative free transform (Danvy and Nielsen TCS 2003)

\[ M \xrightarrow{ad} \mathcal{T}[M] \xrightarrow{ad} P \]
\[ \quad \xleftarrow{ad} \quad \mathcal{T}[N] \xleftarrow{ad} Q \quad \xrightarrow{ad} \quad N \]
Reasoning up to out-of-synch administration

Reason up to bisimulation

\[ M \longrightarrow N \]

\[ \sim \quad \sim \]

\[ P \longrightarrow Q \]

\[ M \sim P \text{ iff } \mathcal{M}[M] \text{ ret } \longrightarrow_{ad} P \]
Reasoning up to out-of-synch aliasing

Reason up to bisimulation

\[ M \sim P \text{ iff } M \equiv \mathcal{N}^{-1}\langle P \rangle \]
Bisimulation technique

Start out similar

\[
\begin{align*}
M \\
\sim \\
\mathcal{T}[M]
\end{align*}
\]

Keep being similar

\[
\begin{align*}
M &\quad \xrightarrow{-\sim} \quad N \\
\sim &\quad \sim \\
P &\quad \xrightarrow{-\sim} \quad Q \\
M &\quad \xrightarrow{-\sim} \quad N \\
\sim &\quad \sim \\
P &\quad \xrightarrow{-\sim} \quad Q
\end{align*}
\]

End up similar

\[
\begin{align*}
M &\downarrow \\
\sim \\
P &\quad \xrightarrow{-\sim} \quad Q \\
M &\quad \xrightarrow{-\sim} \quad N \downarrow \\
\sim \\
Q &\downarrow
\end{align*}
\]
One direction suffices

The forward direction sufficient if (Leroy):

- The source language is deterministic;
- No infinite loop in source terminates in target

Dichotomy of source reductions (Danvy and Zerny, PPDP’13) guarantees point 2:

- Proper reduction *must* cause work in target
- Administrative reduction *must* terminate
Sharing
Call-by-need evaluation

\[
\text{let } x = 1 + 2 \text{ in } x \times x
\]
Call-by-need evaluation

\[
\text{let } x = 1 + 2 \text{ in } x \times x
\]
Call-by-need evaluation

\[
\text{let } x = 1 + 2 \text{ in } x \times x
\]
Call-by-need evaluation

\[
\text{let } x = 1 + 2 \text{ in } x \ast x \\
\quad \text{mapsto} \quad \text{let } x = 3 \text{ in } x \ast x
\]
Call-by-need evaluation

$\text{let } x = 1 + 2 \text{ in } x \times x$

$\mapsto \text{let } x = 3 \text{ in } x \times x$

$\mapsto \text{let } x = 3 \text{ in } 3 \times x$
Call-by-need evaluation

\[
\begin{align*}
\text{let } x &= 1 + 2 \text{ in } x \times x \\
\text{\quad} &\rightarrow \text{ let } x = 3 \text{ in } x \times x \\
\text{\quad} &\rightarrow \text{ let } x = 3 \text{ in } 3 \times x
\end{align*}
\]
Call-by-need evaluation

```
let x = 1 + 2 in x * x
      ⟷ let x = 3 in x * x
      ⟷ let x = 3 in 3 * x
```
Call-by-need evaluation

\[
\text{let } x = 1 + 2 \text{ in } x \times x
\]
\[
\rightarrow \text{ let } x = 3 \text{ in } x \times x
\]
\[
\rightarrow \text{ let } x = 3 \text{ in } 3 \times x
\]
\[
\rightarrow \text{ let } x = 3 \text{ in } 3 \times 3
\]
Call-by-need evaluation

\[ \text{let } x = 1 + 2 \text{ in } x \times x \]
\[ \mapsto \quad \text{let } x = 3 \text{ in } x \times x \]
\[ \mapsto \quad \text{let } x = 3 \text{ in } 3 \times x \]
\[ \mapsto \quad \text{let } x = 3 \text{ in } 3 \times 3 \]
Call-by-need evaluation

\[ \text{let } x = 1 + 2 \text{ in } x \times x \]
\[ \quad \rightarrow \quad \text{let } x = 3 \text{ in } x \times x \]
\[ \quad \rightarrow \quad \text{let } x = 3 \text{ in } 3 \times x \]
\[ \quad \rightarrow \quad \text{let } x = 3 \text{ in } 3 \times 3 \]
\[ \quad \rightarrow \quad \text{let } x = 3 \text{ in } 9 \]
Call-by-need and stateful CPS

Okasaki call-by-need CPS using assignment

\[
\begin{align*}
C[x] & \triangleq \lambda k. x \ k \\
C[\lambda x. M] & \triangleq \lambda k. k \ (\lambda(x, k'). C[M]k') \\
C[MN] & \triangleq \lambda k. C[M](\lambda v. \nu x. \\
& \quad \quad \quad x := \text{memo}_x(N) \ \text{in} \ \nu(x, k)) \\
\text{memo}_x(N) & \triangleq \lambda k. C[N](\lambda w. x := (\lambda k'. k' w) \ \text{in} \ k \ w)
\end{align*}
\]
On liveness of variables

- First evaluate $M$, with $N$ assigned to $x$

$$\textbf{let } x = N \textbf{ in } M$$

- When $x$ is forced, evaluate $N$ and $x$ is no longer in scope

$$\textbf{let } x = N \textbf{ in } E[x]$$

- When $N$ becomes $V$, continue in body with $V$ assigned to $x$

$$\textbf{let } x = V \textbf{ in } E[V]$$
Constructive update

Initial binding is **ephemeral**, disappears on lookup

\[
\nu f. f ::= \_ \in f \ k \\
\quad \mapsto \nu f. \ \text{memo}_f(N) \ k
\]

Updated binding is **permanent**, always available and can never be changed

\[
\nu f. f ::= V \in f \ k \\
\quad \mapsto \nu f. f ::= V \in V \ k
\]
Call-by-need and constructive update CPS

Thunking protocol strictly enforced (thunks only evaluated once):

\[ C[x] \triangleq \lambda k. x \, k \]

\[ C[\lambda x. M] \triangleq \lambda k. k \, (\lambda (x, k'). C[M] \, k') \]

\[ C[MN] \triangleq \lambda k. C[M](\lambda v. v \, x. \ x :=_1 \ \text{memo}_x(N) \ \text{in} \ v(x, k)) \]

\[ \text{memo}_x(N) \triangleq \lambda k. C[N](\lambda w. x := (\lambda k'. k' \, w) \ \text{in} \ k \ w) \]
Constructive update in the $\pi$-calculus

Permanent assignment: replicated server

$$\mathcal{P}[x := \lambda y. M \textbf{in } N] = !x(y). \mathcal{P}[M] \mid \mathcal{P}[N]$$

Ephemeral assignment: unreplicated server

$$\mathcal{P}[x :=_1 \lambda y. M \textbf{in } N] = x(y). \mathcal{P}[M] \mid \mathcal{P}[N]$$

Processes can now responsively change their behavior
Constructive update in machines

- Suspended computations removed on retrieval (Sestoft)
- Thunks become dead when forced
  - “Black holes” in GHC
- Practical implementation techniques reflected by theory
Conclusions

- Program transformation from CPS to processes
- Unite CBN and CBV with CBNeed
- Constructive update model of memoization
- CPS $\lambda$-calculus for change over time
Conclusions

- Program transformation from CPS to processes
- Unite CBN and CBV with CBNeed
- Constructive update model of memoization
- CPS $\lambda$-calculus for change over time
- Reminder: Decisions have consequences; live with them
Questions?